XBC - eXtended Block Cache

Lihu Rappoport

Stephan Jourdan

Yoav Almog

Mattan Erez

Adi Yoaz

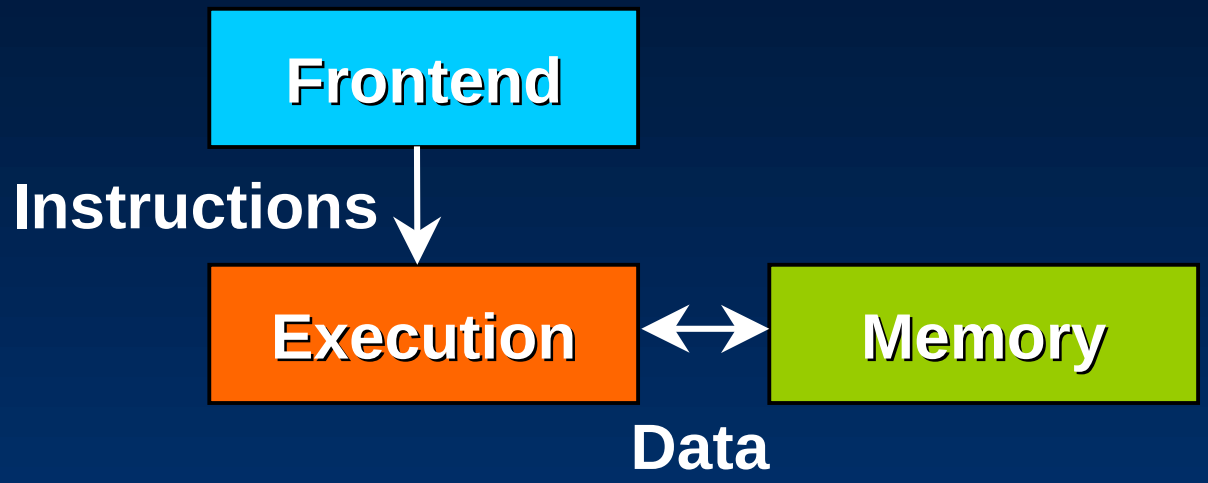
Ronny Ronen

Intel Corporation

intel fi

The Frontend

- The processor:



- Frontend goal: supply instructions to execution
  - Predict which instructions to fetch
  - Fetch the instructions from cache / memory
  - Decode the instructions
  - Deliver the decoded instructions to execution

Requirements from the Frontend

- High bandwidth
- Low latency

The Traditional Solution: Instruction Cache

- **Basic unit: *cache line***

  A sequence of consecutive instructions in memory

- **Deficiencies:**

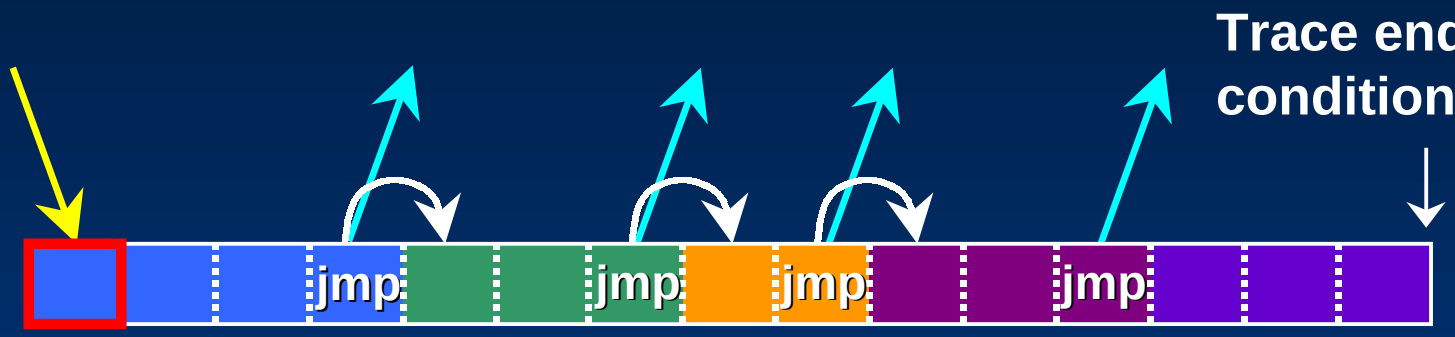
  Low Bandwidth

  High Latency

  Instructions need decoding

Trace Cache

- TC Goals: high bandwidth & low latency
- Basic unit: *trace*
  - A sequence of dynamically executed instructions

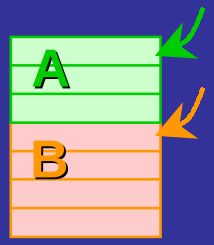


- Instructions are decoded into *uops*
  - Fixed length, RISC like instructions
- Traces have a single entry, and multiple exits
  - Trace tag/index is derived from starting IP

Redundancy in the TC

Code

```
If (cond)
    A
B
```



Possible Traces

(i) AB (ii) B

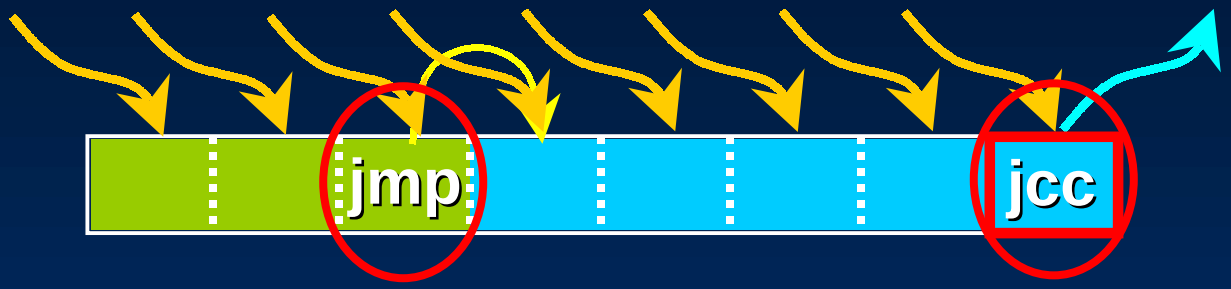
Space inefficiency ⇒ low hit rate

XBC Goals

- High bandwidth
- Low latency
- High hit rate

XBC - eXtended Block Cache

- Basic unit: XB - eXtended Block

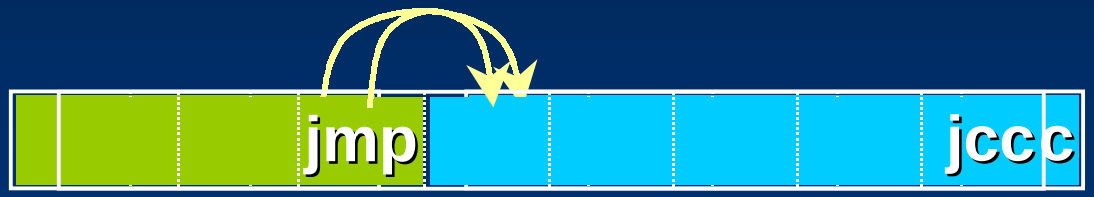


- XB end conditions
  - Conditional or indirect branches
  - Call/Return
  - Quota (16 uops)
- XB features
  - Multiple entry, single exit
  - Tag / index derived from ending instruction IP
  - Instructions are decoded

XBC Fetch Bandwidth

- **Fetch multiple XBs per cycle**
  - **A conditional branch ends a XB**
  - **Need to predict only 1 branch/ XB**
  - **Predicting 2 branch/cyc ⇒ fetch 2 XB/cyc**

- **Promote ≥99% biased conditional branches***



≥ 99% biased

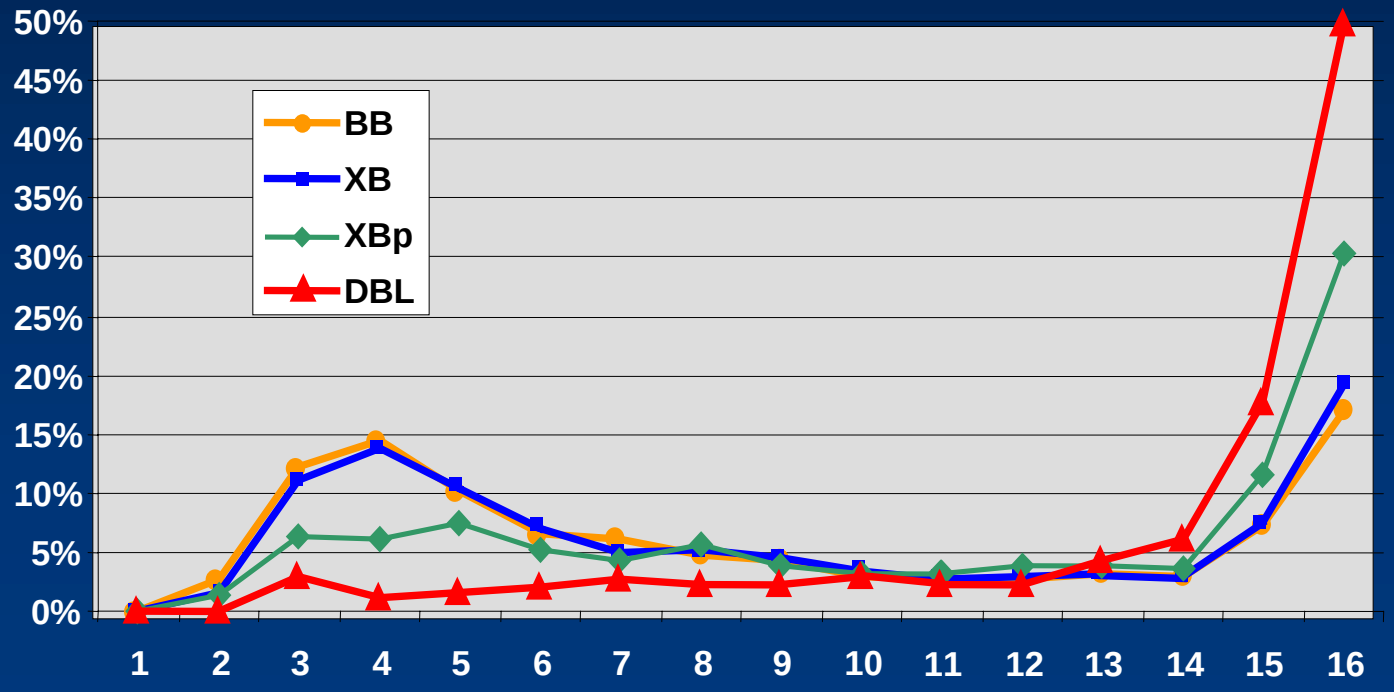
⇒ **Build longer XBs**

⇒ **Maximize XBC bandwidth for a given #pred/cyc**

*[Patel 98]

XB Length

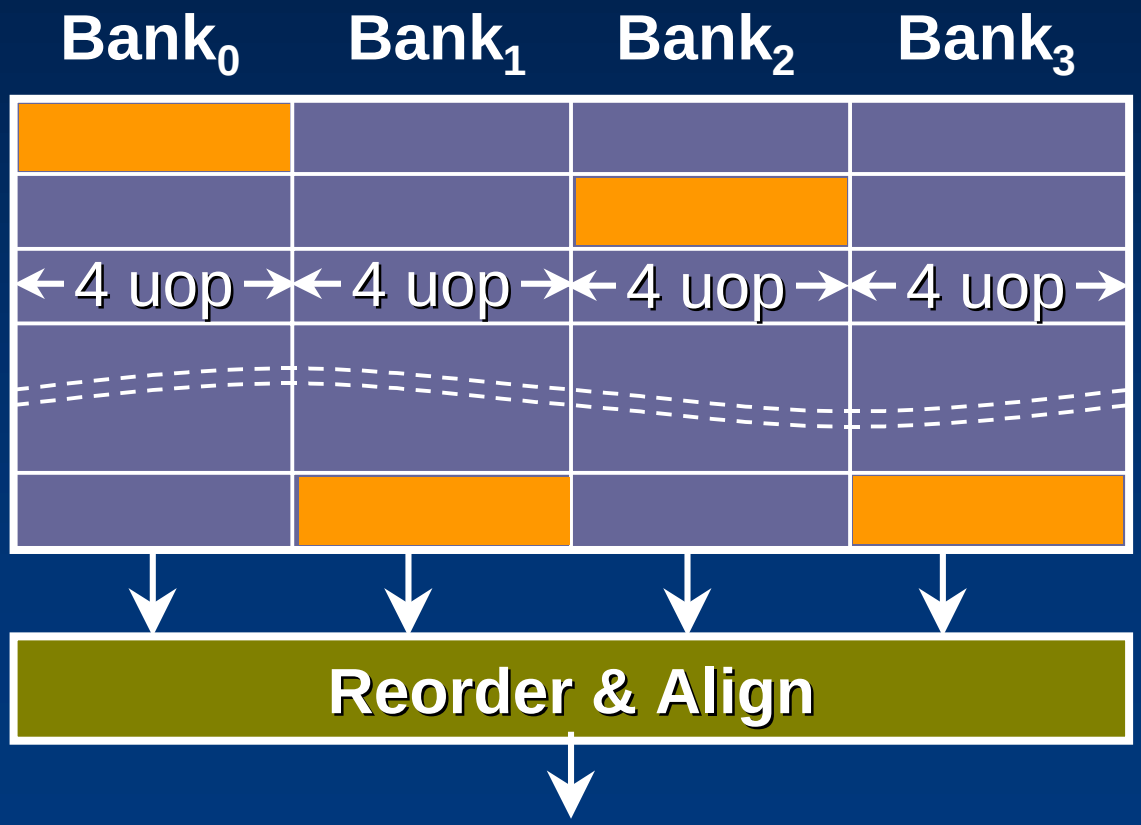
Block types		Average Length
BB	basic block	7.7
XB	don t break on uncond	8.0
XBp	XB + promotion	10.0
DBL	group 2 XBp	12.7



XBC Structure

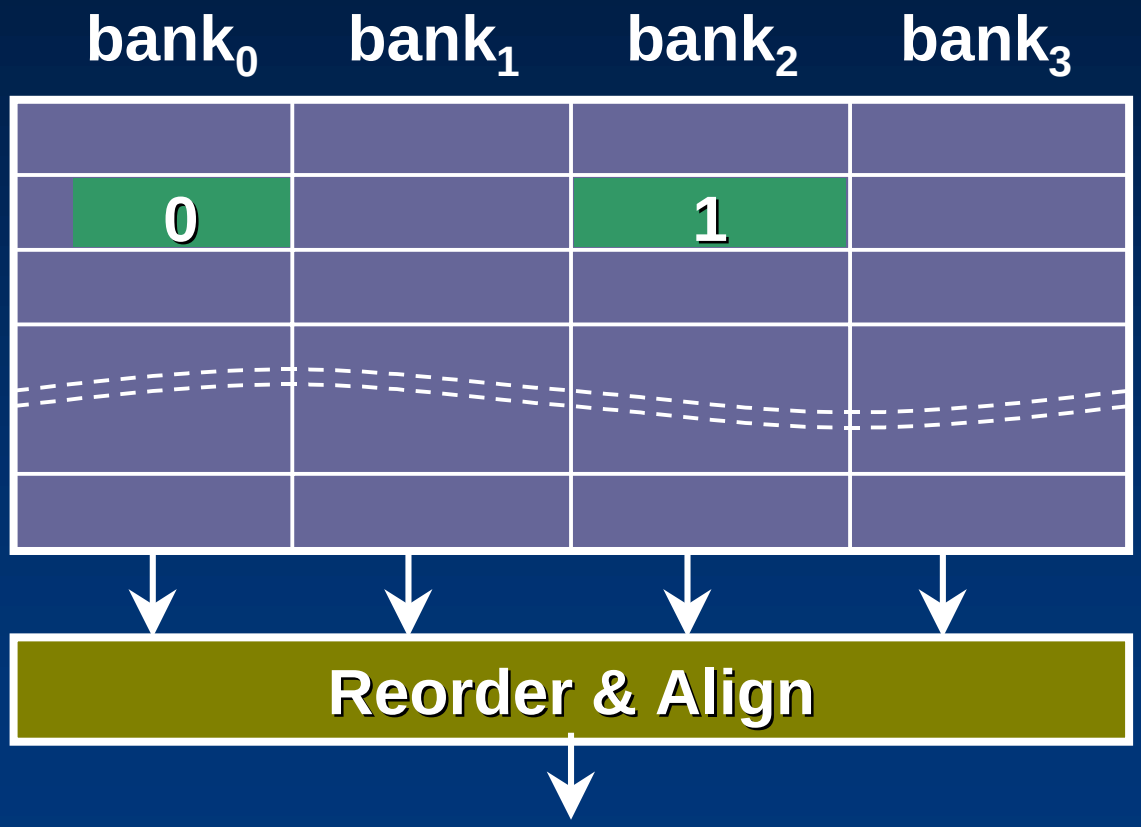
A banked structure which supports

- Variable length XBs (minimize fragmentation)
- Fetching multiple XBs/cycle



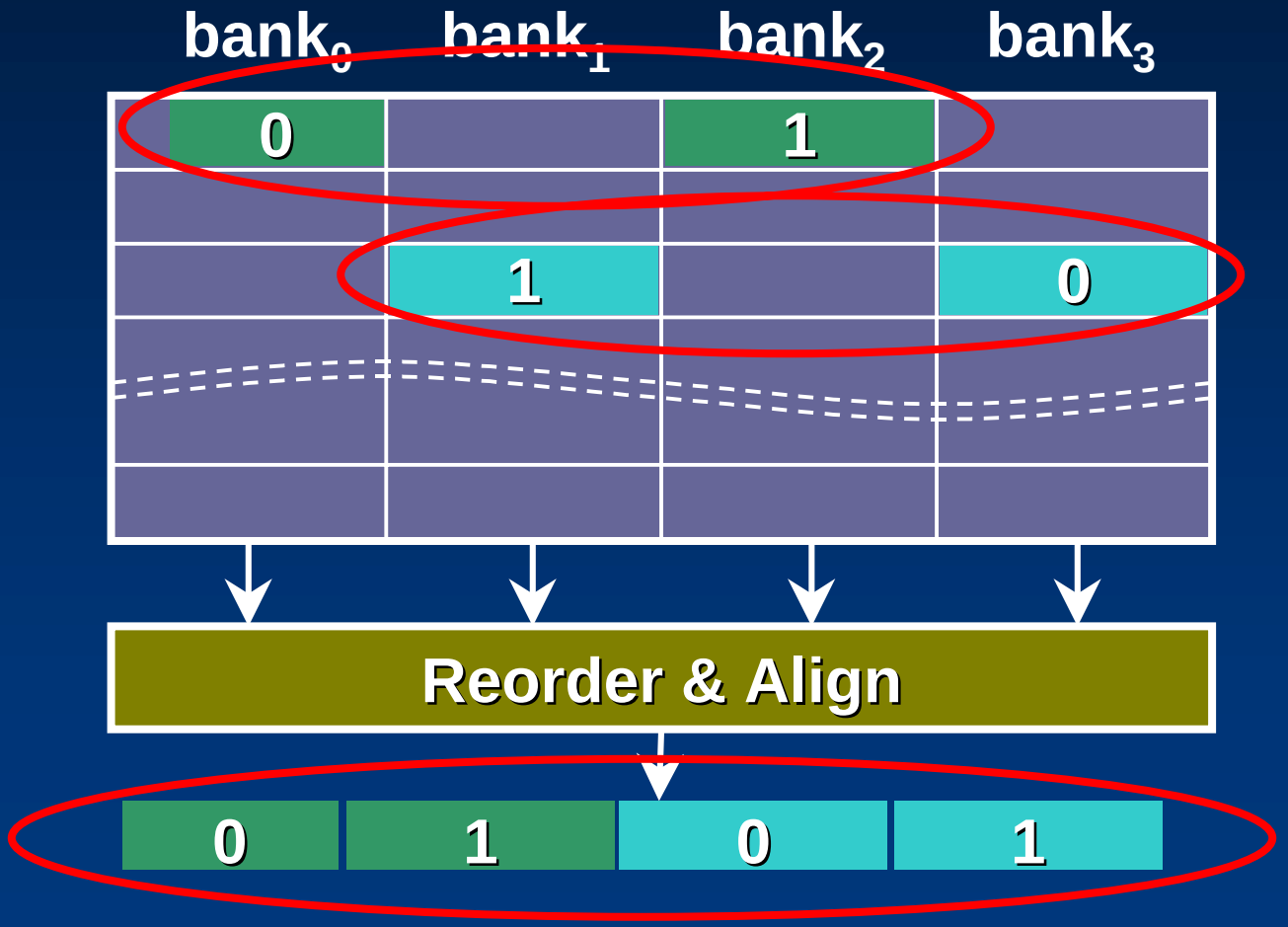
Support Variable Length XBs

- An XB may spread over several Banks on the same set



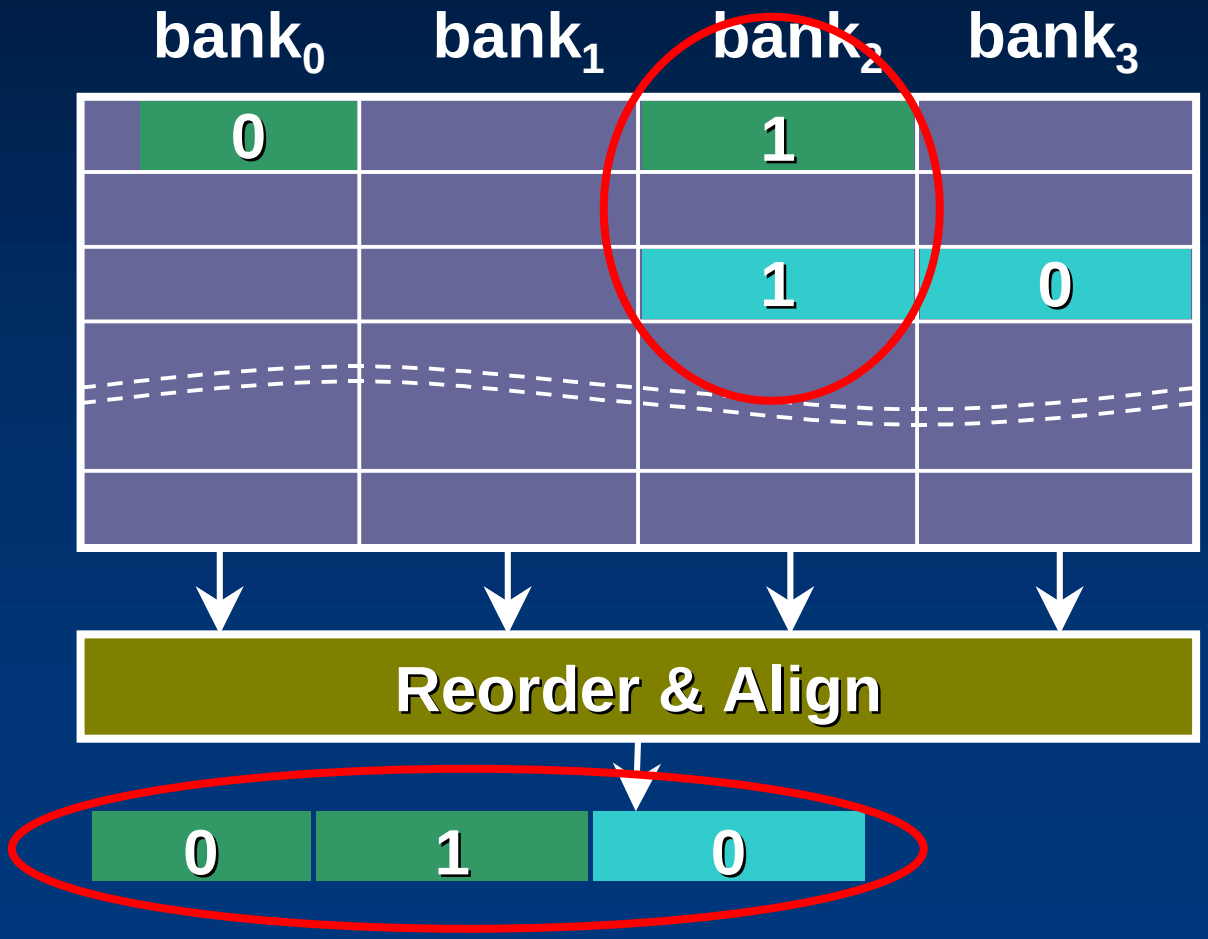
Support Fetching 2 XBs/cycle

- Data may be received from all Banks in the same cycle

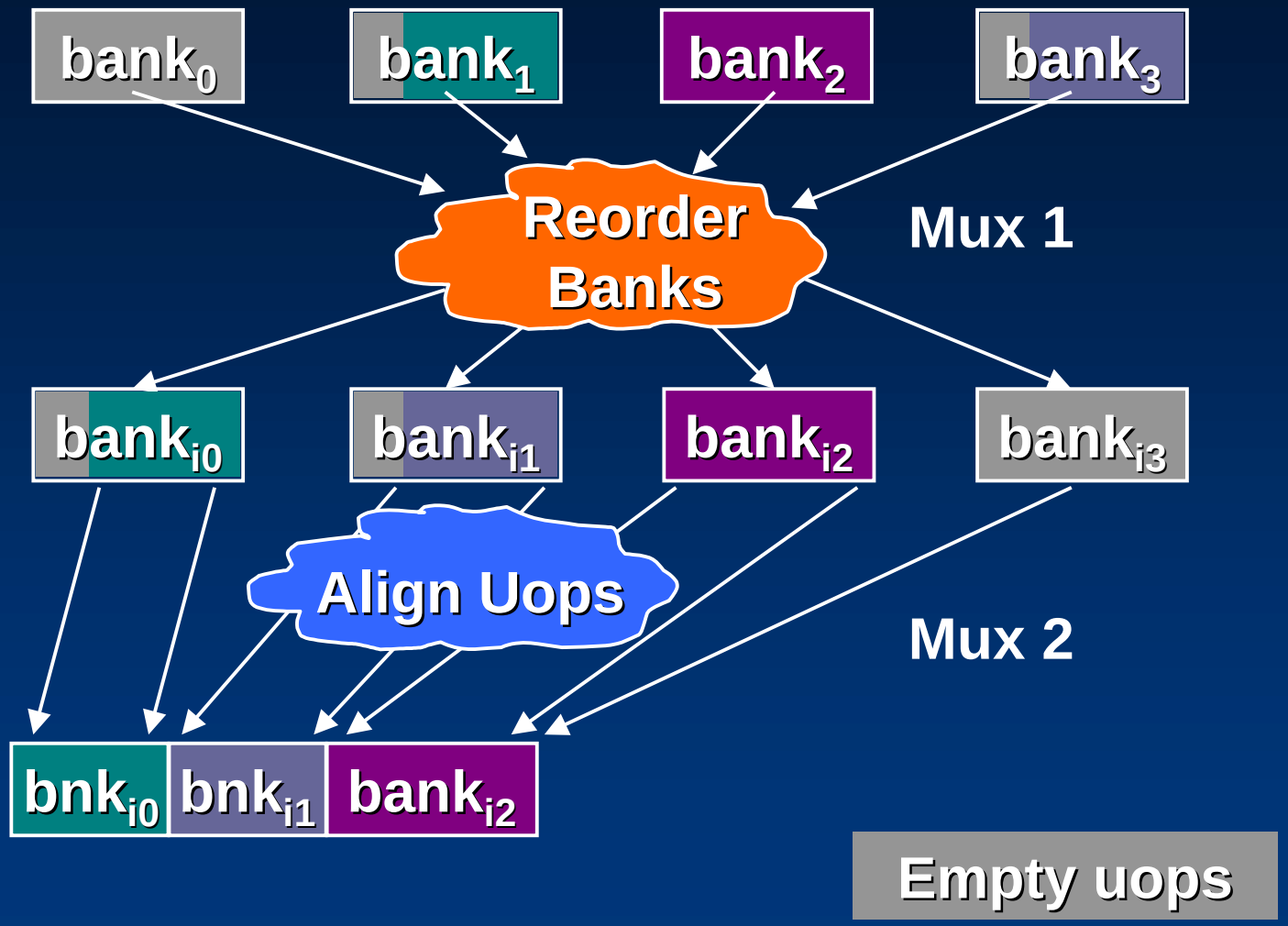


Support Fetching 2 XBs/cycle

- Actual bandwidth may be sometimes less than 4 banks per cycle

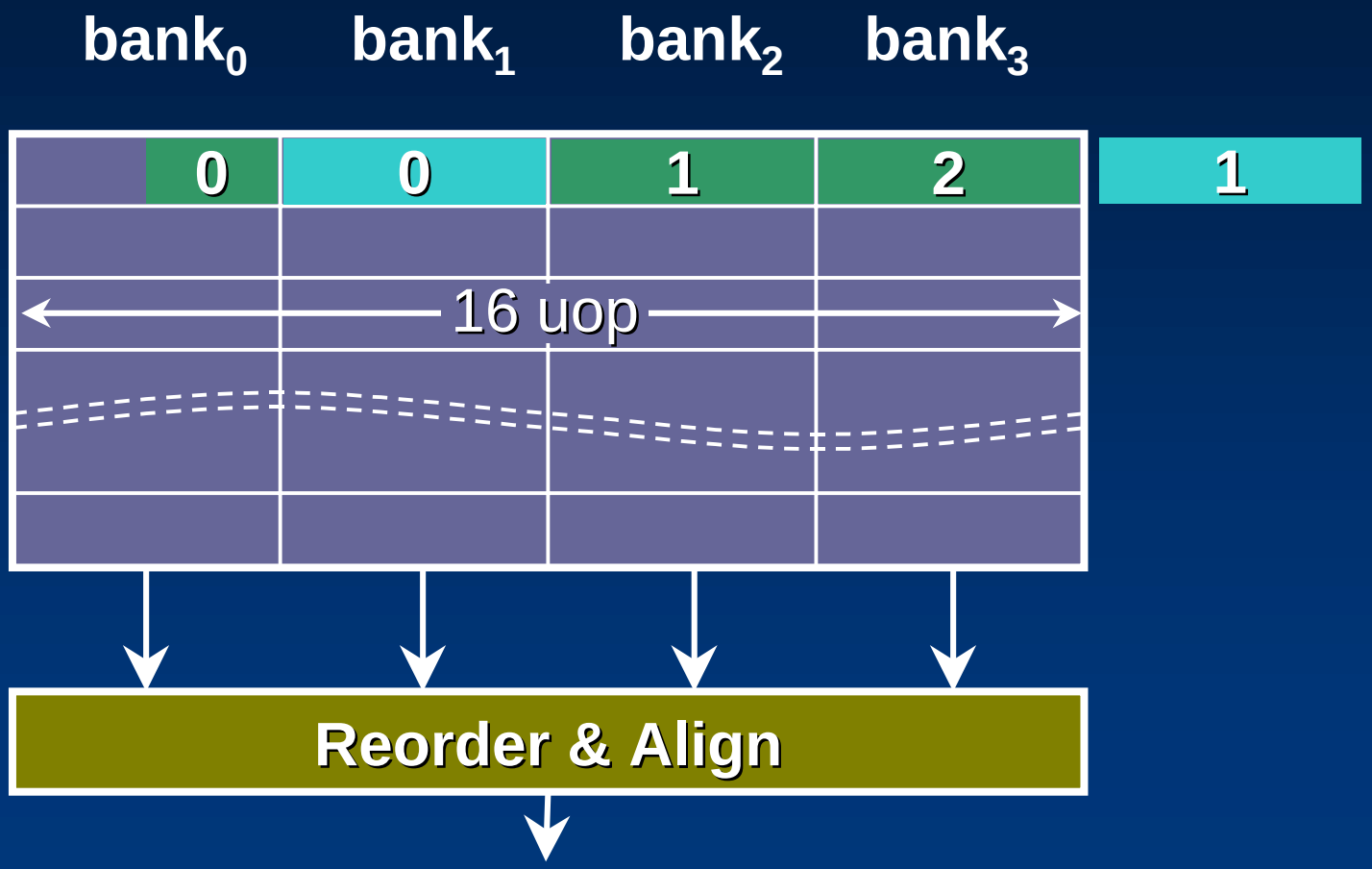


Reordering and Aligning Uops



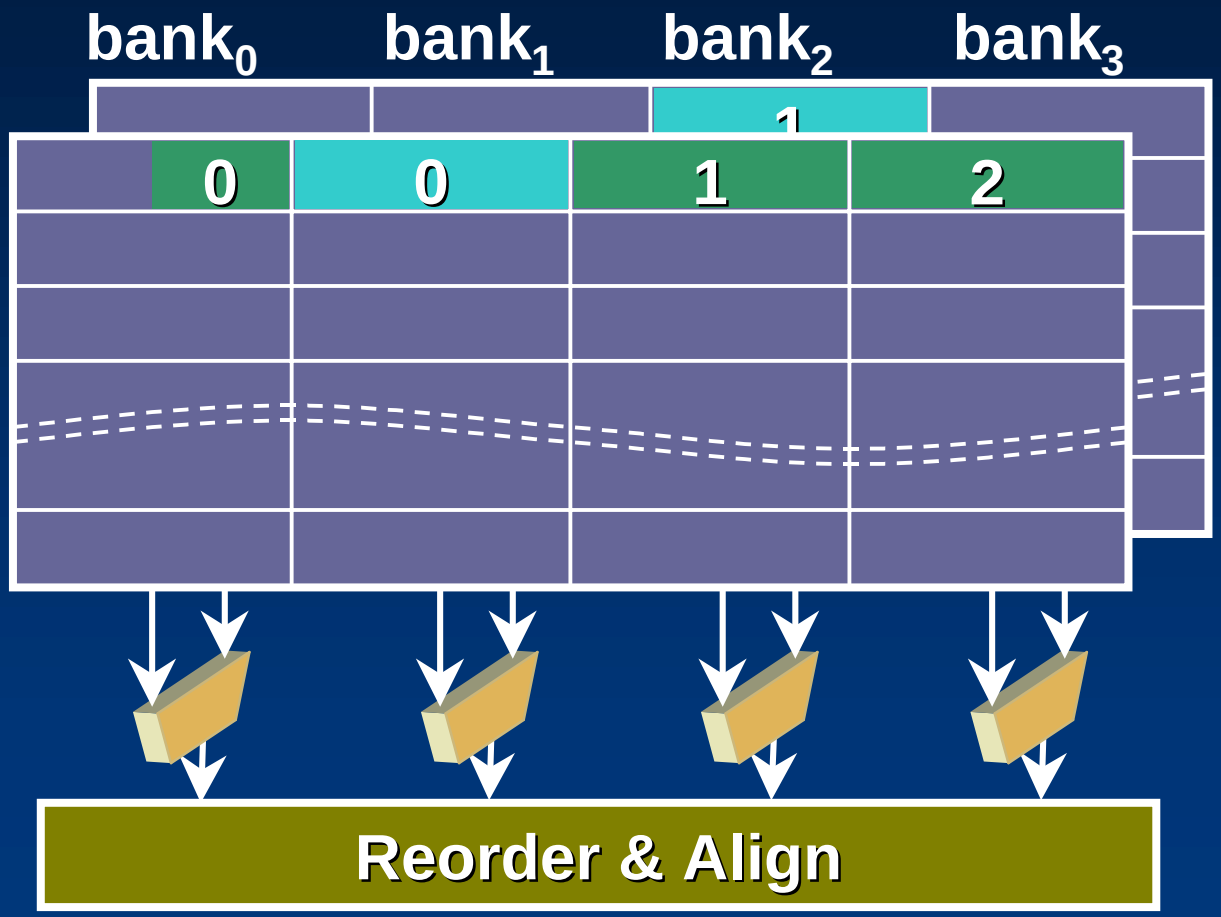
XBC Structure

- The average XB length is >8 uops
  - ⇒ **16 uop/line is < 2-XB set associative**



XBC Structure

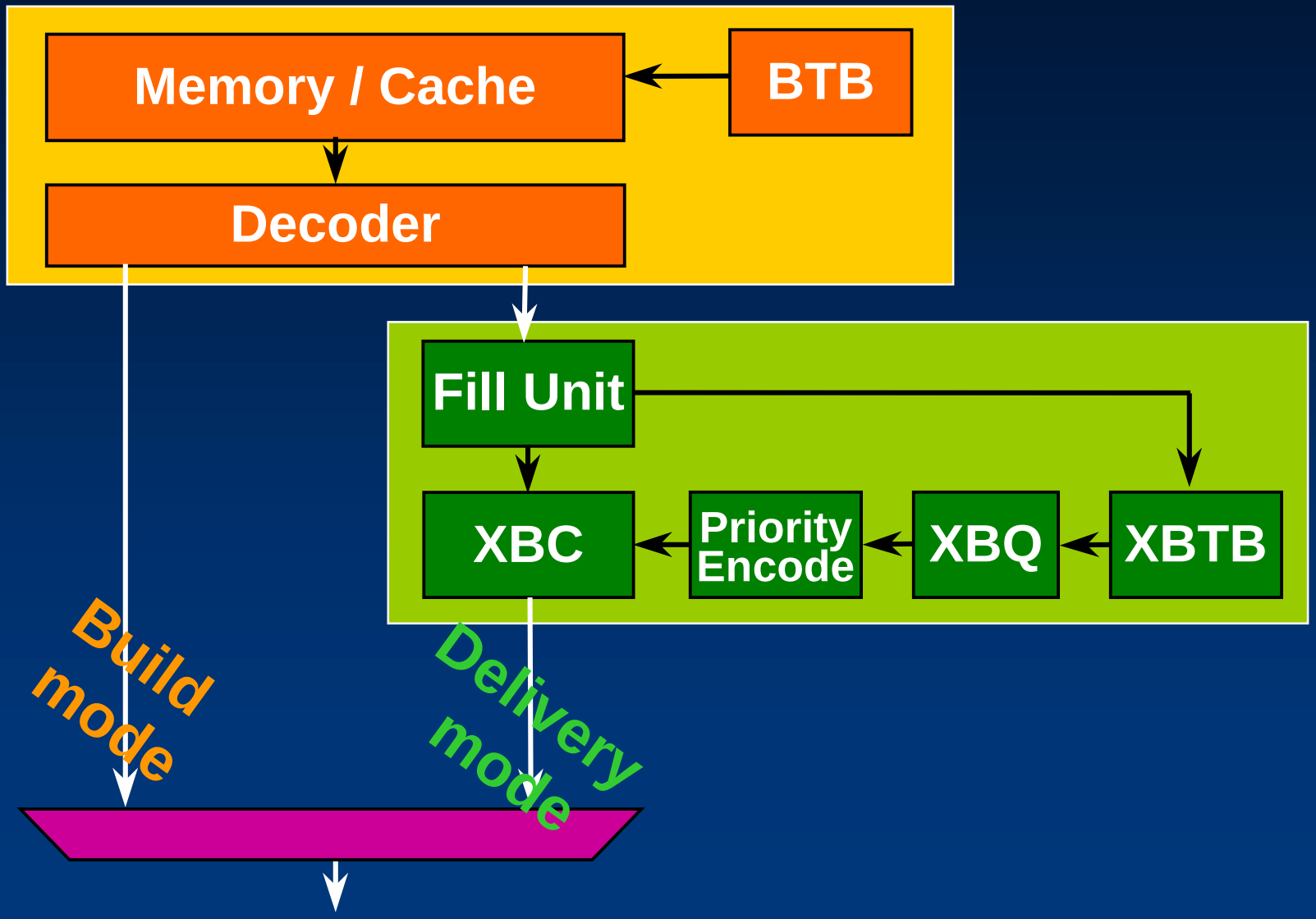
- The average XB length is >8 uops
  ⇒**make each bank set-associative**



The XBTB

- **The XBTB provides the next XB for each XB**
  - **XBs are indexed according to ending IP**
  - **⇒ Cannot directly lookup next IP in the XBC**
  - **⇒ XBC can only be accessed using the XBTB**
- **XBTB provides info needed to access next XB**
  - **The IP of the next XB**
    - Defines the set in which the XB resides
  - **A masking vector, indicating the banks in which the XB resides**
  - **The #uops counted backward from the end of XB**
    - Defines where to enter the XB
- **XBTB provides next 2 XBs**

XBC Structure: the whole picture



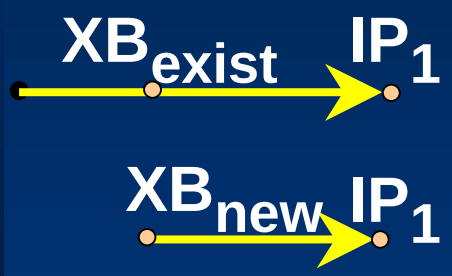
XB Build Algorithm

- XBTB lookup fails ⇒ build a new XB into the fill buffer
- End-of-XB condition reached ⇒ lookup XBC for the new XB

  No match ⇒ store new XB in the XBC, and update XBTB

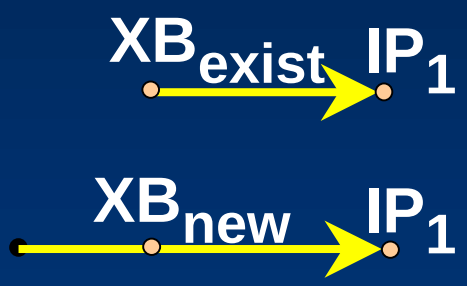
  Match ⇒ there are three cases:

$XB_{new} \subseteq XB_{exist}$



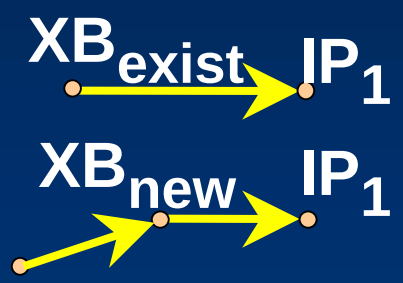
Update XBTB

$XB_{new} \supset XB_{exist}$



Extend XB_{exist}
Update XBTB

$XB_{new} \cap XB_{exist} \neq \varnothing$



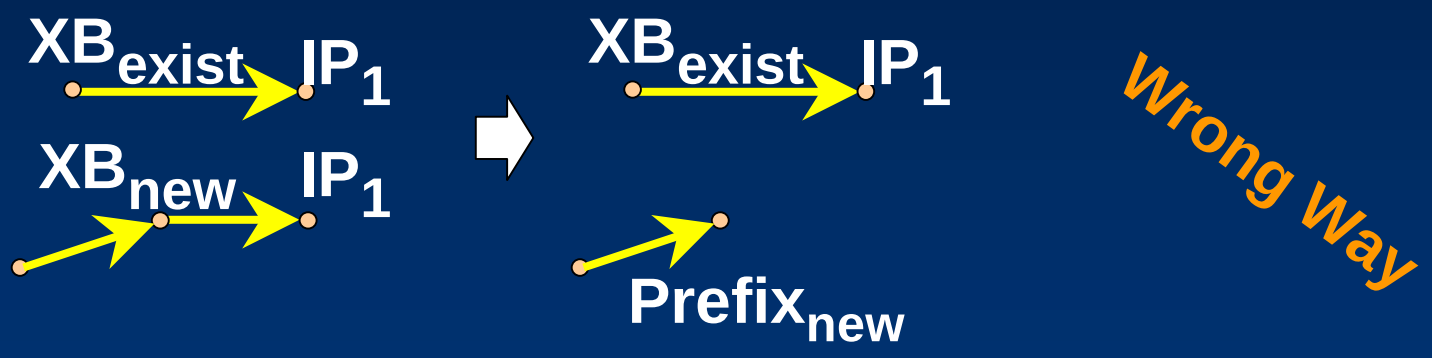
Complex XB,
Update XBTB

The XBC has NO Redundancy

Complex XBs

- XB_{new} and XB_{exist} have same suffix but different prefix:

  Possible solution, complying to no-redundancy:

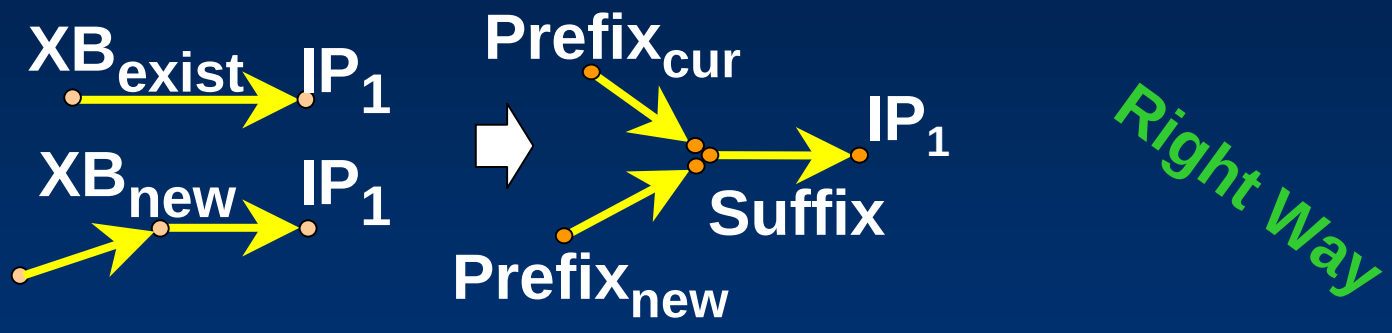


  Drawback: we get 2 short XBs instead of a single long XB

Complex XBs

- XB_{new} and XB_{exist} have same suffix but different prefix:

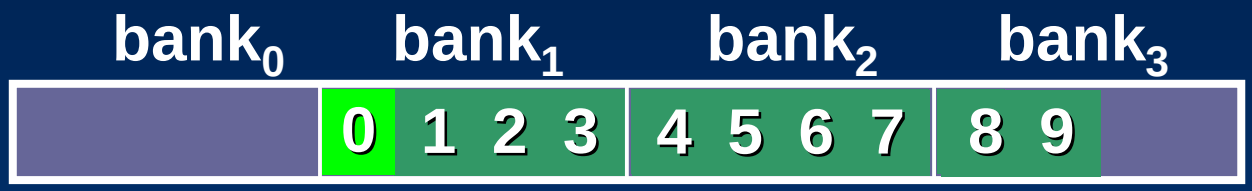
  Second solution: a single complex XB



- Complex XBs: no redundancy, but still high bandwidth

Extending an Existing XB

- An XB can only be extended at its beginning
- If we store XB in the usual way, when an XB is extended, we need to move all its uops



- Since the existing uops move, the pointers in the XBTB become stale

Storing Uops in Reverse Order

- The solution is to store the uops of an XB in a reversed order

⇒ when an XB is extended, no need to move uops

- XB IP is the IP of the ending instruction ⇒ extending the XB does not change the XB IP

Set Search

- **XB is replaced and then placed again**
  - **Not on same set ⇒ different XB**
  - **Same set, same banks ⇒ no problem**
  - **Same set but not on the same banks ⇒**

    XBTB entries which point to the old location of the XB are erroneous
- **Solution - Set Search**
  - **On an XBTB hit & XBC miss, try to locate the XB in other banks in the same set**
  - **Calculate new mask according to offset**
  - **Only a small penalty: cycle loss, but no switch to build**

XB Replacement

- Use a LRU among all the lines in a given set
- LRU also makes sure that we do not evict a line other than the first line of a XB (a head line)
  - There is no point in retaining the head line while evicting another line
    - if we enter the XB in the head line, we will get a miss when we reach the evicted line
    - if a head line is evicted, but we enter the XB in its middle, we may still avoid a miss
- A non-head line is always accessed after a head line is accessed

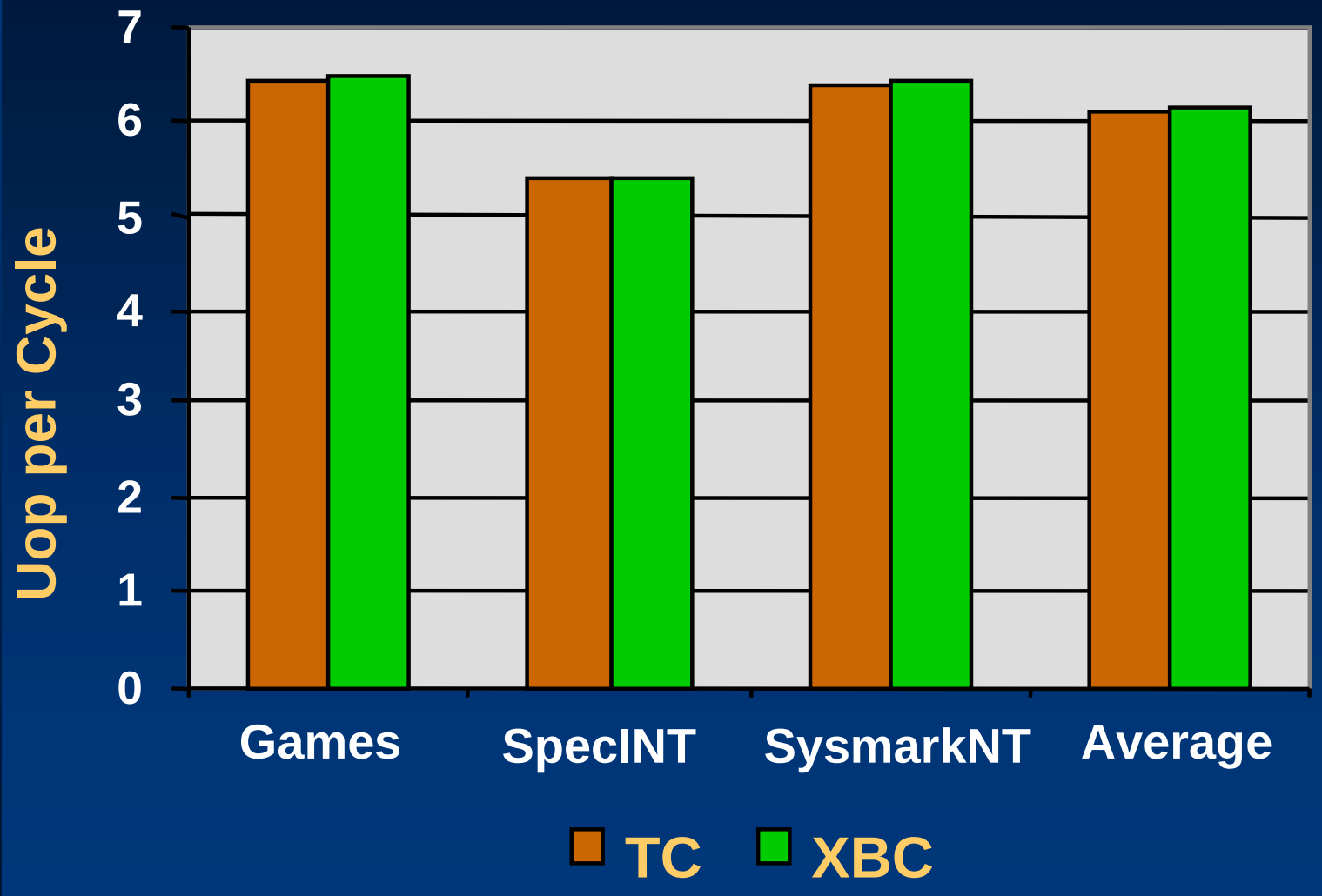
  $\Rightarrow$ its LRU will be higher

  $\Rightarrow$ it will not be evicted before the head line

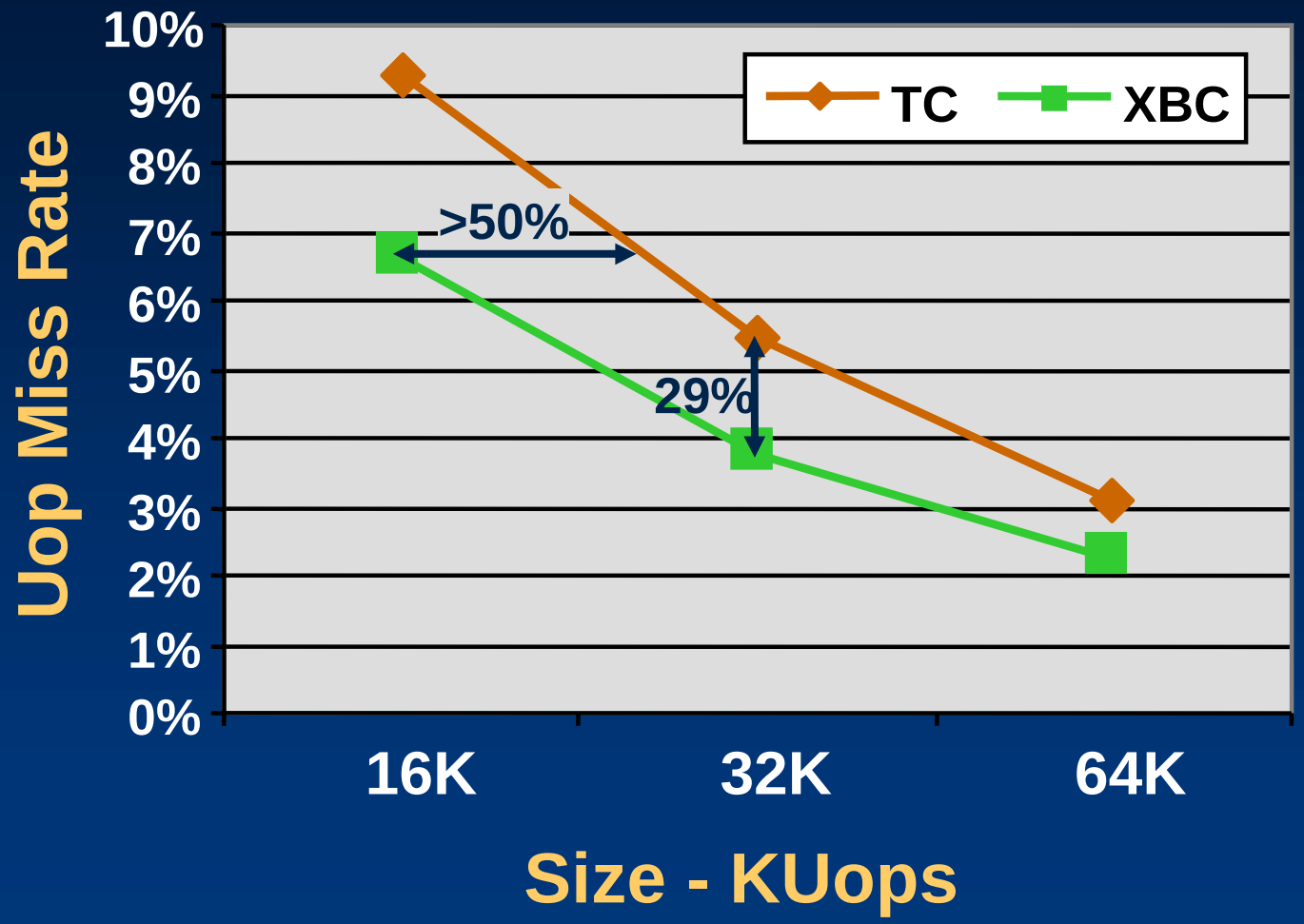
XB Placement

- **Build-mode placement algorithm**
  - New XB is placed in banks such that it does not have bank conflict with the previous XB (if possible)
  - LRU ordering is maintained by switching the LRU line with the non-conflicting line before the new XB is placed
  - Set-search repairs the XBTB
- **Delivery mode placement algorithm**
  - repeating bandwidth losses due to bank conflicts found
    - ⇒ conflicting lines are moved to non-conflicting banks
  - Each XB is augmented with a counter
    - incremented when XB has a bank conflict
    - when counter reaches threshold, the conflicting lines are switched with other lines in non-conflicting banks
  - A line can be switched with another line, only if its LRU is higher, or if both gain from the switch

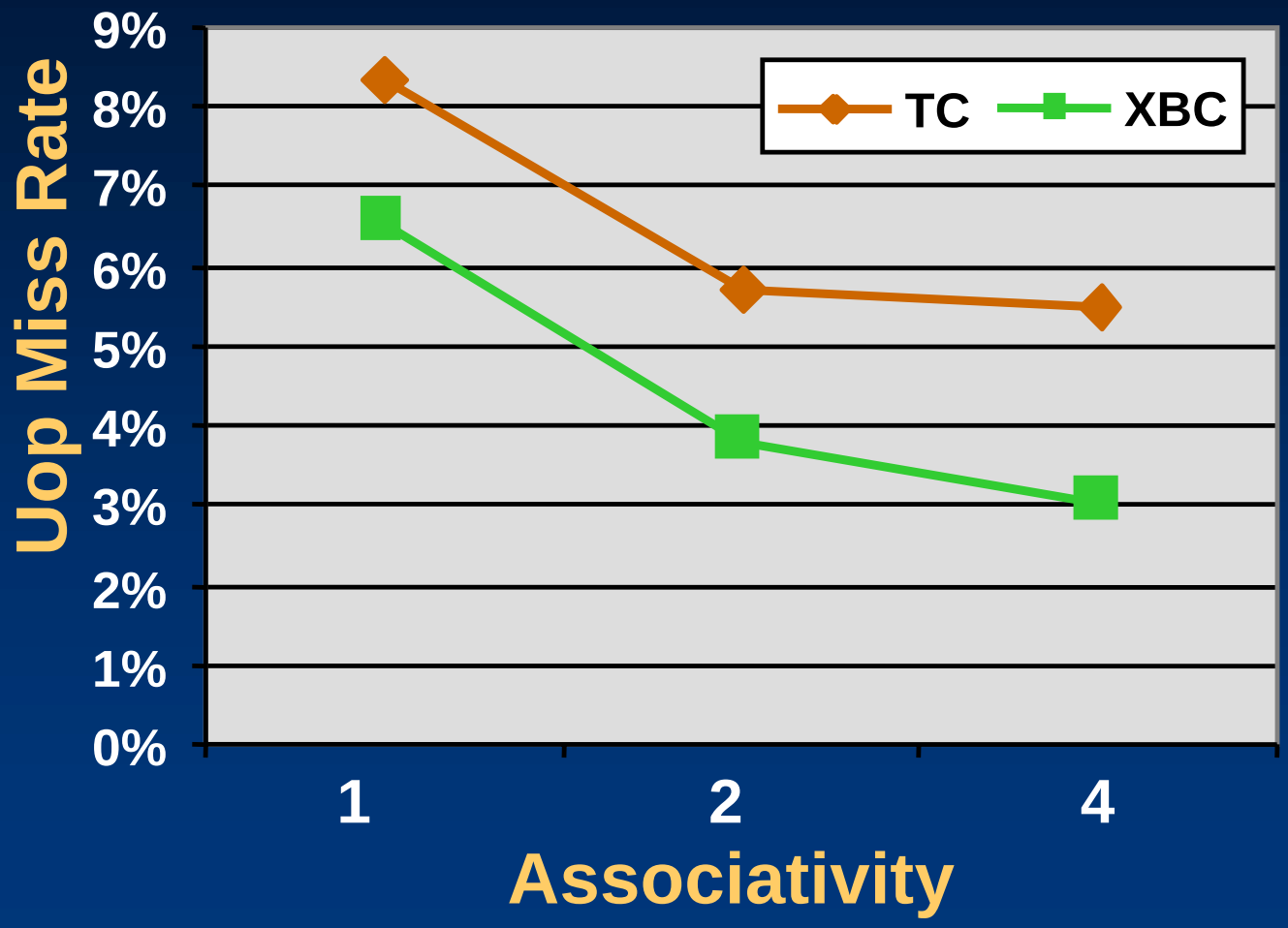
XBC vs. TC Delivery Bandwidth



Miss Rate as a Function of Size



Miss Rate as a Function of Size



XBC Features Summary

- Basic unit - XB
  - Ends with a conditional branch
  - Multiple entries, single exit
  - Indexed according to ending IP
  - Branch promotion ⇒ longer XBs
- XBC uses a banked structure
  - Supports fetching multiple XBs/cycle
  - Supports variable length XBs
  - Uops within XBs are stored in reverse order

Conclusions

- Instruction Cache has high hit rate, but
  Low bandwidth, high latency
- TC has high bandwidth, low latency, but
  Low hit rate
- XBC combines the best of both worlds
  High bandwidth, low latency and high hit rate

End of Talk?

Almost .

intel fi

Hit Rate, Switches...

- Same Bandwidth, Same latency, higher hit-rate

  Is it always better?

- No!
- In IA32: IC/TC latencies differ
  Switches to build mode costs!
- Example:

  TC: 5uop BW, ~70% hit

  IC: 2uop BW, ~30% hit

  Fetching 35 inst =
  35*70%/5+35*30%/5= 5+5 = 10 clks

  Perfect branch prediction
- Single switch: 10+3=13 cycle

- 5 switches: 10+3*5 = 25 cycles